Towards the construction of distributed detection programs, with an application to distributed termination

Jean-Michel Hélary, Michel Raynal

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1460

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION

Jean-Michel HÉLARY
Michel RAYNAL

Juin 1991

RR-1460

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX
FRANCE
Téléphone : 99.36.20.00
Télex : UNIRISA 950 473F
Télécopie : 99.38.38.32

Publication Interne n° 585 - Mai 1991 - 24 Pages
Programme 3

Towards the construction of distributed detection programs, with an application to distributed termination*

Jean-Michel Hélary, Michel Raynal
IRISA , Campus de Beaulieu, 35042 RENNES CEDEX, FRANCE
FAX 99 383832, Telex UNIRISA 950 473F
e-mail helary@irisa.fr,raynal@irisa.fr

Vers la construction de programmes de détection répartie; application à la terminaison répartie.

Abstract

Methodological design of distributed programs is of major concern to master parallelism. Due to their role in distributed systems, the class of observation or detection programs, whose aim is to observe or detect properties of an observed program, is very important. The detection of a property generally rests upon consistent evaluations of a predicate; such a predicate can be global, i.e. involve states of several processes and channels of the observed program. Unfortunately, in a distributed system, the consistency of an evaluation cannot be trivially obtained. This is a central problem in distributed evaluations. This paper addresses the problem of distributed evaluation, used as a basic tool for solution of general distributed detection problems. A new evaluation paradigm is put forward, and a general distributed detection program is designed, introducing the iterative scheme of *guarded waves sequence*. The case of distributed termination detection is then taken to illustrate the proposed methodological design.

*This work was supported by French Research Program C^3 on Parallelism and Distributed Computing

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(U.R.A. 227)
UNIVERSITE DE RENNES I I.N.S.A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(UNITE DE RECHERCHE DE RENNES)

Résumé

La conception méthodique de programmes répartis est d'une grande importance dans la maîtrise du parallélisme. A cause de leur rôle dans les systèmes répartis, la classe des programmes d'observation ou de détection, dont le rôle est d'observer ou de détecter des propriétés d'autres programmes répartis, est fondamentale. La détection d'une propriété repose généralement sur l'évaluation cohérente d'un prédicat; ce dernier peut être global, c'est-à-dire dépendre de l'état de plusieurs des processus et canaux du programme observé. Malheureusement, dans un environnement réparti, la cohérence d'une évaluation ne s'obtient pas de manière triviale. Ce problème est au centre de l'évaluation répartie. Dans ce rapport, on examine le problème de l'évaluation répartie, utilisée comme outil de base pour résoudre le problème de la détection répartie. Un nouveau paradigme d'évaluation est mis en évidence, puis un programme général de détection est conçu, sur la base d'un nouveau schéma itératif, la *séquence de vagues guardées*. Le cas de la terminaison sert ensuite à illustrer la méthode générale de conception.

1 Introduction

Studying a program can be made by analyzing the set of its possible runs. These runs can be characterized by properties, defined over their states. In an asynchronous distributed context, programs are composed of processes and channels, and it may be very difficult to detect such properties since it is impossible to observe at the same time all the components of a program. It is well-known that this impossibility is the core of the difficulties encountered in the control of distributed programs. Specific solutions have been formulated for particuliar properties, specially in the case of stable ones, e.g. termination or deadlock detection [4, 5, 6, 8, 12, 13, 17], or for properties whose detection lie over the record of a global snapshot [2, 7].

In this paper, we are primarly interested in the problem of evaluating a predicate, defined over the states of a distributed program, whether this predicate is stable or not. Informally, we have a distributed predicate (or even a distributed function of a more general type) whose values depend on variables belonging to different processes sharing no common memory. Each component of the predicate, depending on a single process, can be evaluated by that process in a single state; however, each process having its own control flow, all these partial evaluations cannot be performed in the same state: that's why a global evaluation is necessarily *non-simultaneous*. Moreover, values obtained on each process have to be collected, in order to get complete evaluation. Since this collect is not instantaneous, the final result of the evaluation is *delayed*. These informal notions are formalized with the important paradigm of *Non Simultaneous Delayed Evaluation* (NSDE). This concept catches the notion of *distributed snapshot of local evaluations*. Such a NSDE is worked out by a control program: the notion of *superimposition*, previously made clear by several authors [1, 3], is a useful tool to modelize this task.

The second goal of this paper is to contribute to the development of methods for distributed detection of properties, expressed by a predicates over the states of a distributed program. An iterative scheme, based upon a sequence of NSDEs, is well suited to this development. We generalize to the distributed context the usual iteration deriva-

tion techniques known in a sequential context, namely the expression of a result as the conjunction of an invariant and a stop condition. Invariant condition is expressed as a conjunction of local predicates (*guards*), and can be insured by subjecting the progression of each NSDE to the satisfaction of these guards. On the other hand, the stop condition is evaluated by each NSDE: the value returned determines whether another round has to be performed, or whether the expected result has been obtained. This leads to the second original concept put forward in this paper, namely the *guarded waves sequence*. Together with this construction, safety and liveness properties are addressed very carefully.

The rest of the paper is organized as follows: in §2, we present the computational model, based upon interleaving semantics, and making use of what we call *behaviour rules* for events. Evaluation problems are addressed in §3, details of construction for the detection method are in §4. In the conclusion, we outline some possible applications to similar detection problems, involving *monotonic* functions.

2 The computationnal model

A distributed program is made of a network of n processes $P_1, P_2, \ldots, P_n$, sharing no common memory. They communicate only by exchanging messages through communication channels. Let $X = \{1, \ldots, n\}$ be the set of process identifiers, and $\Gamma \subseteq X \times X$ the set of channels. Each channel is an ordered pair $c_{ij} = (i, j)$ carrying messages from P_i to P_j. The directed graph $G = (X, \Gamma)$, modelizing the network, is supposed to be *strongly connected*: there is a directed path from any i to any j ($i \in X, j \in X$). Each process is a sequential program, with its own context. Messages carry values, since they are used to exchange information between processes. Thus a message is made of two parts: a *header*, containing such informations as message identification, sender and/or receiver identities, etc. and a *value* (of a certain type). The two parts of a message m will be denoted respectively by $m.id$ and $m.v$. For the ease of exposition, we will suppose that each message is uniquely identified.

To a distributed program is associated a set of *events*, which correspond to the executions of atomic operations performed by the processes. There are three types of events: *sends, receipts* and *internal events*. Each send or receive event is parameterized by a single message. We will denote by

$E_{ij}(m)$ the event "P_i sends message m on channel c_{ij}",

$R_{ij}(m)$ the event "P_j receives message m on channel c_{ij}",

INT_i an internal event of P_i.

To each message m sent over a channel c_{ij} are associated the two predicates $sent(i, j, m)$ and $rec(i, j, m)$ expressing respectively the facts that m has been sent by P_i on c_{ij} (resp. received by P_j on c_{ij}). When a process P_i performs a send event $E_{ij}(m)$, it assigns a value to $m.v$. On the other hand, when a process P_j performs a receive event $R_{ij}(m)$, the value $m.v$ is assigned to a local variable of P_j.

To represent a *run* of a distributed program, we use the well-known *event-state interleaving* model. The run proceeds in a sequence of steps. Each step is a *triple* (s, e, s') where s ans s' belong to a set of *states* and e is an event: when the program is in state s and the event e occurs, the next state is s'. Recall that the state of the program is the set of values of all variables (including those representing channels). A step (s, e, s')

is called a *transition* from state s to state s'. Moreover, to each event e is associated an *enabling condition* $en(e)$, which is a state assertion. The notation $\{p\}\ e\ \{q\}$, where p and q are state assertions, means, on the one hand, that $en(e) \Rightarrow p$ and, on the other hand, that for the step (s, e, s'), if p holds in state s, then q holds in state s'. Such a notation will be called a *behaviour rule*.

For instance, $\forall m,\ \forall (i,j) \in \Gamma$:

$\{\neg sent(i,j,m)\}\ E_{ij}(m)\ \{sent(i,j,m)\}$
$\{\neg rec(i,j,m)\}\ R_{ij}(m)\ \{rec(i,j,m)\}$

When assertion $en(e)$ is true in a state s and remains true as long as e doesn't occur, we say that e is *forever enabled from the state s.*

A run of a distributed program is thus a *sequence* of steps $s_0\ e_0\ s_1 \ldots s_k\ e_k\ s_{k+1} \ldots$ such that, for all $k \geq 0$, s_k is a state, e_k is enabled in state s_k and s_{k+1} is the state resulting when e_k occurs in state s_k. It will be assumed that distinct events are never simultaneous, so that each run corresponds to a totally ordered list of states. We will use the following notations: in a given run,

- $s < s'$ iff state s is before state s' in the run (s is earlier than s')
- given a finite set of states $\mathcal{S} = \{s^{(1)}, \ldots, s^{(p)}\}$ belonging to a run, $\max(s^{(1)}, \ldots, s^{(p)})$ (resp. min) is the latest (resp. earliest) state from this set:

$$\max(s^{(1)}, \ldots, s^{(p)}) \in \mathcal{S} \text{ and } 1 \leq j \leq p \Rightarrow s^{(j)} \leq \max(s^{(1)}, \ldots, s^{(p)})$$

To each event e can be associated an *occurrence predicate* $p(e)$ defined in the following way: if e occurs in a step (s, e, s'), then $p(e)$ holds in a state σ if, and only if, $s' \leq \sigma$. Finally, following Lamport, we will say that event e *leads to* event e' if any occurrence of e is necessarily followed by an occurrence of e' after a finite number of steps (denoted by $e \overset{*}{\Rightarrow} e'$).

This model allows us to give a precise meaning to the notions of *progress, non-determinism* and *fairness*, and to the hypotheses of *asynchronous* and *reliable* communications. For a given run, the following properties can be defined:

progress : when at least one event is enabled in state s, one of these events occurs in state s,

non-determinism : when several events are enabled in state s, any one of them may occur in state s,

fairness : when an event is forever enabled from a state s, it will eventually occur (after a finite number of steps).

Also, we will say that communication is *asynchronous* if, for any channel c_{ij} and any message m on c_{ij}, the number of steps between the two events $E_{ij}(m)$ and $R_{ij}(m)$ is unpredictable. Thus, in any run of an asynchronous program, we have:

$$\forall m,\ \forall (i,j) \in \Gamma :\ en(R_{ij}(m)) \Rightarrow sent(i,j,m) \wedge \neg rec(i,j,m)$$

We say that a channel is *reliable* iff, for any message m on channel c_{ij}:

1. $sent(i,j,m)$ is the occurrence predicate of $E_{ij}(m)$ and $rec(m,i,j)$ is the occurrence predicate of $R_{ij}(m)$. This implies:
   (a) channel c_{ij} cannot spontaneously create messages ($sent(i,j,m)$ cannot hold unless event $E_{ij}(m)$ occurred),
   (b) channel c_{ij} cannot duplicate messages ($rec(i,j,m)$ remains true forever after $R_{ij}(m)$ occurred),
   (c) channel c_{ij} cannot loose messages ($sent(i,j,m)$ remains true forever after $E_{ij}(m)$ occurred and $rec(i,j,m)$ cannot hold unless event $R_{ij}(m)$ occurred)).
2. $R_{ij}(m)$ is forever enabled after a finite number of state transitions after the occurrence of $E_{ij}(m)$.

From this it follows that, if channels are reliable and run is fair, every send event $E_{ij}(m)$ leads to the corresponding receipt $R_{ij}(m)$, and no receipt $R_{ij}(m)$ can occur unless a corresponding $E_{ij}(m)$ occurred before (recall that we assume messages are uniquely identified).

From now on, we will restrict ourselves to this model of computation, with asynchronous and reliable communications. We will consider only distributed programs whose runs verify progress, non-determinism and fairness assumptions. All subsequent notions will be relative to an arbitrary run of a distributed program, unless otherwise stated by the use of quantifiers over the set of possible runs, such as *for any run ...* or *there exists a run such that...*

3 Evaluating predicates

3.1 Some concepts about distributed evaluation

Consider a function F defined over the set of states of a distributed program. The value of F in state s will be denoted by $F[s]$.The evaluation of F by a process P in a state s is an operation which allows this process to compute $F[s]$.

The evaluation of F by a process P is *atomic* (or instantaneous) if no state transition occurs during this evaluation. In an asynchronous distributed model, atomic evaluation of F by P is possible if, and only if, F depends only of P's local context since, in such a computation model, no process can have an immediate access to other processes context: this is inherent to the model.

For example, any predicate $sent(i,j,m)$ can be atomically evaluated by P_i but not by P_j. Similarly, $rec(i,j,m)$ can be atomically evaluated by P_j but not by P_i.

Several functions $F_1,\ldots,F_k$ can be ***simultaneously evaluated*** by a process P if there exists a state s such that P can atomically evaluate $F_1[s],\ldots,F_k[s]$. Like in the case of atomic evaluation, simultaneous evaluation is possible in asynchronous distributed model if, and only if, functions to be evaluated depend only of variables all belonging to the same process.

Let $\mathcal{F} = f(F_1,F_2,\ldots,F_n)$ be a function such that F_i is a function atomically evaluable by P_i.

Definition 3.1. A non-simultaneous evaluation of $\mathcal{F}$ is an evaluation $f(F_1[s^{(1)}], \ldots, F_n[s^{(n)}])$, where $s^{(i)}$ denotes the state in which P_i evaluates F_i.

Definition 3.2. A non-simultaneous delayed evaluation (*NSDE*) of $\mathcal{F}$ by process P is a non simultaneous evaluation whose result is recorded in P's local context in a state $s \geq \max(s^{(1)}, \ldots, s^{(n)})$. Such an evaluation will be denoted by

$$\mathcal{F}[s^{(1)}, \ldots, s^{(n)} \mid s]$$

This concept captures the essence of "global" function evaluation in asynchronous distributed systems. The following definitions give a more precise meaning to the quality of such evaluations in the case of boolean functions (predicates).

Definition 3.3 A *NSDE* of $\mathcal{F}$ is said to be *safe* with regard to a predicate $\mathcal{T}$ whenever

$$\mathcal{F}[s^{(1)}, \ldots, s^{(n)} \mid s] \Rightarrow \mathcal{T}[s]$$

Definition 3.4 A predicate $\mathcal{F}$ is said to be *live* with regard to predicate $\mathcal{T}$ whenever, in any run,

$$\mathcal{T}[s] \Rightarrow \exists (s^{(1)}, \ldots, s^{(n)}, s') \text{ with}$$

$$\left(s \leq \min(s^{(1)}, \ldots, s^{(n)}) \leq \max(s^{(1)}, \ldots, s^{(n)}) \leq s'\right) \wedge \mathcal{F}[s^{(1)}, \ldots, s^{(n)} \mid s']$$

Note Definition 3.3 is weaker than the assertion $\mathcal{F} \Rightarrow \mathcal{T}$ since the latter means that implication holds in all states of the program, while our definition is related to a particuliar NSDE. Similarly, if we consider, as in [3], the relation (over predicates) $\mathcal{T}$ *ensures* $\mathcal{F}$, which means that, if $\mathcal{T}[s]$ holds, then $\mathcal{F}$ will eventually hold in a state $\sigma \geq s$, liveness definition 3.4 is weaker, since it doesn't assert the existence of a state σ in which an atomic evaluation $\mathcal{F}[\sigma]$ would give the result *true*, but only the existence of a NSDE whose result is *true*. The safety and liveness notions stated here are thus closely related to the concept of NSDE. In that sense, they are different, for instance, from the *detects* relation in UNITY [3].

3.2 Control program for evaluation: the superimposition model

Consider a distributed program, called *underlying program*, and a predicate $\mathcal{F}$ defined over the states of this program. Performing a NSDE of $\mathcal{F}$ is a task which may involve some messages, variables, and thus events, not belonging to the underlying program. This task is thus worked out by a program different from the underlying program, which is called a *control program* (relative to the underlying program). This notion is well-known; a formal framework well-suited to its expression is the ***superimposition model*** [1, 3], which can be described as follows:

- To each proces P_i of the underlying program is associated a process C_i (controller) of the control program,

- A non-empty subset of the local context of each P_i can be atomically "observed" by the corresponding controller C_i: this means that all functions defined over this subset can be atomically evaluated by C_i,
- To each event of the underlying program corresponds an event of the control program. In particular, the control program simulates emission and receipt of the underlying program messages: to each *send* and *receive* event of the underlying program corresponds a *send* and *receive* event of the control program, with the same message parameter.
- In addition to underlying events, the control program has specific events (*control events*). For instance, sends or receipts of messages not belonging to the underlying program, called *control messages*, are such control events.

Thus, any run of the superimposed control program is partially dependant of a run of the underlying program. The local context of each controller C_i comprises:

1. P_i's variables atomically observable by C_i: their value depend only of the run of the underlying program (P_i can read and write it, but C_i can only *read* them)
2. perhaps some other variables (control variables), specific to the control program (they can be *read* and *written* by C_i, and are hidden to P_i).

Processes C_i can communicate through a set of channels including Γ (the underlying program set of channels) and possibly extra channels (which can be used only by control messages). Finally, to each state of the underlying program corresponds a state of the control program: a run of the underlying program is a subsequence of the corresponding run of the control program. Designing a superimposed control program consists in specifying control events and their associated behaviour rules.

In what follows, UM will denote the set of underlying messages.

3.3 Designing a control program for NSDE

Recall that a NSDE of predicate $\mathcal{F} = f(F_1, \ldots, F_n)$ by a process C is obtained when a set of values $F_1[s^{(1)}], \ldots, F_n[s^{(n)}]$ has been recorded (in a state s) in the context of C. Two events, *start* and *return* correspond to the beginning and the end of the NSDE program. To which process belongs *start* is not specified at this level of abstraction, whereas *return* belongs to the process recording the result of NSDE. For each $i \in X$ let *collected*(i) be a predicate, expressing the fact that process C_i has participated in the NSDE of $\mathcal{F}$, in other words, has evaluated F_i. The NSDE is specified by the following behaviour rules:

$$\begin{array}{lll} \{en(start)\} & start & \{\bigwedge_{i \in X} (\neg collected(i))\} \\ \{\bigwedge_{i \in X} (collected(i))\} & return & \{a = \mathcal{F}[s^{(1)}, \ldots, s^{(n)} \mid s]\} \end{array}$$

where a denotes the value returned by NSDE.

In order to progress from the post-condition of *start* to the precondition of *return*, each process C_i must perform an atomic evaluation of F_i in a state $s^{(i)}$. This evaluation corresponds to an event, namely $visit_i$. This event must occur once and only once during the step, so we obtain the following behaviour rules:

$$\forall i \in X \quad \begin{array}{lll} \{en(start)\} & start & \{\bigwedge_{i\in X}(\neg collected(i))\} \\ \{\neg collected(i)\} & visit_i & \{collected(i)\} \\ \{\bigwedge_{i\in X}(collected(i))\} & return & \{a = \mathcal{F}[s^{(1)},\ldots,s^{(n)} \mid s]\} \end{array}$$

Finally, we will say that NSDE program is *live* whenever, in any run, $start \stackrel{*}{\Rightarrow} return$.

This set of behaviour rules is an abstract specification of the now classical tool known as a *wave* [9, 15, 16]. In the next section, we use and generalize this tool in order to obtain a detection control program.

4 Detection problems

Let $\mathcal{T}$ be a predicate defined over the states of a distributed program. We want to solve the following problem: *detect a state s of the distributed program such that* $\mathcal{T}[s] = true$.

We don't assume that $\mathcal{T}$ is stable (recall that a predicate $\mathcal{P}$ is said to be *stable* if, and only if: $(\mathcal{P}[s] \wedge s' \geq s) \Rightarrow \mathcal{P}[s']$).

4.1 Derivation of a detection control program

Let $\mathcal{F}$ be a predicate such that it is possible to perform a sequence ($k = 0, 1 \ldots$) of NSDEs

$$\mathcal{F}[s_k^{(1)},\ldots,s_k^{(n)} \mid s_k] \text{ with :}$$

$$\forall k \geq 0 \ : \ \max(s_k^{(1)},\ldots,s_k^{(n)}) \leq s_k \leq \min(s_{k+1}^{(1)},\ldots,s_{k+1}^{(n)}) \qquad (SC)$$

The predicate $\mathcal{F}$ will solve the detection problem stated above if it meets the two requirements:

1. detection must be safe, that is to say:

   $\forall k \geq 0$, the k^{th} NSDE is safe with respect to $\mathcal{T}$. This insures that, whenever exists k such that $\mathcal{F}[s_k^{(1)},\ldots,s_k^{(n)} \mid s_k]$ is true, we can conclude $\mathcal{T}[s_k]$ is also true.

2. The detection must be live, that is to say:

   $$\mathcal{T}[s] \Rightarrow \exists k \ : \ s_k \geq s \wedge \mathcal{F}[s_k^{(1)},\ldots,s_k^{(n)} \mid s_k]$$

   This insures that, whenever exists s such that $\mathcal{T}[s]$ is true, then it exists k such that the k^{th} NSDE will return true.

Now, suppose we can separate $\mathcal{F}$ into two parts $I, \mathcal{A}$ verifying

$$\forall k \geq 0 \ : \ I \equiv \bigwedge_{i\in X} I_i[s_k^{(i)}]$$

where I_i is a predicate atomically evaluable by C_i ($i \in X$), and $\mathcal{A} = f(A_1,\ldots,A_n)$ (each A_i atomically evaluable by C_i), such that

$$I \wedge \mathcal{A}[s_k^{(1)},\ldots,s_k^{(n)} \mid s_k] \equiv \mathcal{F}[s_k^{(1)},\ldots,s_k^{(n)} \mid s_k]$$

I is a *loop invariant* and $\mathcal{A}$ is the associated *stop condition*. The sequence of NSDEs must be designed in order that, for all $k \geq 0$, states $s_k^{(i)}, i \in X$ and s_k verify:

(1) $\mathcal{A}[s_k^{(1)}, \ldots, s_k^{(n)} \mid s_k]$ is evaluated (let a_k denote this value),

(2) $\bigwedge_{i \in X} I_i[s_k^{(i)}]$ is true,

These two conditions are relative to each iteration step. We omit subscript k, since we refer to a single iteration step. Condition **(1)** is realized with a *wave* as seen in §3.3. But we must also insure the loop invariant **(2)**. To this end, each state $s^{(i)}$ must be such that $I_i[s^{(i)}]$ be true. In fact, it may happen that, according to the underlying program and to the implementation of the wave, some i exist for which I_i is false in the state where $visit_i$ occurs. To overcome this, each event $visit_i$ is no longer considered as atomic: it is split into two distinct events, namely beg_visit_i, end_visit_i. The former corresponds to the occurrence of C_i's visit, according to the wave, the latter is enabled as soon as I_i becomes true: I_i is the *guard* of the event end_visit_i. For each $i \in X$, the predicate $wh(i)$ expresses the fact that event beg_visit_i occurred, but event end_visit_i didn't yet occur (for the current step). Denoting by ss, $sb^{(i)}$, $se^{(i)}$, sr the states in which events *start*, beg_visit_i, end_visit_i, *return* respectively occur (indexed by the number k of the current wave if necessary), we obtain the behaviour rules: (predicates $wh(i), i \in X$ are assumed to be initially false)

$\{en(start)\}$	*start*	$\{\bigwedge_{i \in X} (\neg collected(i) \wedge \neg wh(i))\}$
$\{\neg wh(i) \wedge \neg collected(i)\}$	beg_visit_i	$\{wh(i) \wedge \neg collected(i)\}$
$\{wh(i) \wedge \neg collected(i) \wedge I_i\}$	end_visit_i	$\{\neg wh(i) \wedge collected(i)\}$
$\{\bigwedge_{i \in X} (I_i[se^{(i)}] \wedge collected(i))\}$	*return*	$\{a = \mathcal{A}[se^{(1)}, \ldots, se^{(n)} \mid sr]\}$

Hence, the loop invariant (or wave guard) $I \equiv \bigwedge_{i \in X} I_i[se^{(i)}]$ holds.

Now, the sequentiality condition (SC) has to be satisfied. For that purpose, we define $en(start)$ and strenghten the postcondition of *return*; a predicate *new_step* expresses the fact that a new step can start ($en(start) \equiv new\_step$ and *new_step* is assumed initially true):

$\{new\_step\}$	*start*	$\{\neg new\_step \wedge \bigwedge_{i \in X} (\neg collected(i) \wedge \neg wh(i))\}$
$\{\neg wh(i) \wedge \neg collected(i)\}$	beg_visit_i	$\{wh(i) \wedge \neg collected(i)\}$
$\{wh(i) \wedge \neg collected(i) \wedge I_i\}$	end_visit_i	$\{\neg wh(i) \wedge collected(i)\}$
$\{\bigwedge_{i \in X} (I_i[se^{(i)}] \wedge collected(i))\}$	*return*	$\{a = \mathcal{A}[se^{(1)}, \ldots, se^{(n)} \mid sr] \wedge new\_step = \neg a\}$

This set of behaviour rules corresponds to what can be called a *sequence of guarded waves*. It generalizes the well-known "sequence of waves" scheme, in the sense that, for each wave, each event end_visit_i is preconditionned (guarded) by an assertion I_i.

4.2 Safety and liveness

Let's address the question of how to meet safety and liveness requirements.

Safety. To meet this requirement, it is sufficient to find guards I_i and stop condition $\mathcal{A}$ such that, in any run,

$$\bigwedge_{i \in X} \left(I_i[se^{(i)}] \right) \wedge \mathcal{A}[se^{(1)}, \ldots, se^{(n)} \mid sr] \Rightarrow \mathcal{T}[sr]$$

Liveness. This point is not so obvious. First, it is necessary that waves themselves are live, that is to say, in any run:

$$(l1) \qquad \forall i \in X \ : \ start \ \stackrel{\bullet}{\Rightarrow} \ beg\_visit_i$$
$$(l2) \quad (\forall i \in X \ : \ end\_visit_i) \ \stackrel{\bullet}{\Rightarrow} \ return$$

These requirements depend only on the implementation of waves, which is a part of the control program design, and not of the underlying program.

If waves are live, we can assert that, in any state of the control program, either there is a "current wave" or the stop condition has been met in a preceeding state. More precisely, in any state s of the control program, there is an integer $k \geq 1$ such that, one and only one of the three situations holds:

either $ss_k \leq s \leq sr_k$ (wave k is currently run in state s),

or $sr_{k-1} \leq s \leq ss_k \wedge new\_step[sr_{k-1}]$ (wave $k-1$ has returned and wave k is enabled but not yet started. Event $start_k$ will eventually occur),

or $sr_{k-1} \leq s \wedge \mathcal{A}[se^{(1)}_{k-1}, \ldots, se^{(n)}_{k-1} \mid sr_{k-1}]$ (Wave $k-1$ has returned and stop condition holds. This wave was the last one).

In the first two cases, wave k is the *current* one, and $s \leq sr_k$ holds.

However, waves liveness properties $(l1)$ and $(l2)$ are not sufficient to insure that a wave will eventually return. The following property $(l3)$: "in any run of the control program, $\forall i \in X \ : \ beg\_visit_i \stackrel{\bullet}{\Rightarrow} end\_visit_i$" must also be verified. But $(l3)$ depends on the guards I_i which, in turn, depend on events belonging to the underlying program. Thus, the event end_visit_i might be not forever enabled: even in a fair run, it might happen that beg_visit_i doesn't lead to end_visit_i, and consequently that the current wave never returns. So, in the context of detection, ensuring liveness requires that, if $\mathcal{T}[s]$ is true, then $\exists k$ such that:

1. wave k eventually returns, and
2. $\mathcal{A}[se^{(1)}_k, \ldots, se^{(n)}_k \mid sr_k] \wedge \bigwedge_{i \in X} I_i[se^{(i)}_k]$, with $s \leq \min\left(se^{(1)}_k, \ldots, se^{(n)}_k, sr_k\right)$.

The following lemma gives a sufficient condition, depending on the underlying program behaviour, to obtain a live detection.

Lemma 4.1 If waves are live, and if guards I_i and stop condition $\mathcal{A}$ verify, in any run of the control program:

i) $\mathcal{T}[s] \Rightarrow (\forall i \in X \ \exists \sigma^{(i)} \ : \ (\sigma \geq \sigma^{(i)} \Rightarrow I_i[\sigma]))$

ii) $\mathcal{T}[s] \Rightarrow (s \leq \min(s^{(1)}, \ldots, s^{(n)}) \leq \max(s^{(1)}, \ldots, s^{(n)}) \leq \overline{s} \Rightarrow \mathcal{A}[s^{(1)}, \ldots, s^{(n)} \mid \overline{s}])$

then the detection is live.

Unformally, assumption i) means that, if $\mathcal{T}$ holds in a state s, all the guards I_i eventually hold and are stable. This assumption is used to prove that the current wave eventually returns. Assumption ii) means that every NSDE of the stop predicate $\mathcal{A}$, performed after s, will return the value *true*. This assumption is used to prove that there will be a "last" wave.

Proof Consider an arbitrary run of the control program, for which $\mathcal{T}[s]$ holds.

Suppose the stop predicate holds in state s. In that case, the result is obvious: if k is the number of last wave, then

$$sr_k \leq s \wedge \mathcal{A}[se_k^{(1)}, \ldots, se_k^{(n)} \mid sr_k] \wedge \bigwedge_{i \in X} I_i[se_k^{(i)}]$$

Otherwise, let k be the number of the current wave in state s. From assumption i), $\forall i \in X$, guard I_i continuously holds from state σ_i, whence event end_visit_i is forever enabled from state σ_i. Thus, wave k will eventually return. Two situations are to consider:

a) $s \leq \min(se_k^{(1)}, \ldots, se_k^{(n)})$. In that case, from assumption ii), assertion $\mathcal{A}[se_k^{(1)}, \ldots, se_k^{(n)} \mid sr_k]$ holds, and stop condition is met by wave k, which will be the last one.

b) $s > \min(se_k^{(1)}, \ldots, se_k^{(n)})$. In that case, it may be possible that $\neg\mathcal{A}[se_k^{(1)}, \ldots, se_k^{(n)} \mid sr_k]$. But this means that $new\_step[sr_k]$ holds, and thus wave $k+1$ will eventually start. From assumption i), guards I_i still hold, and thus wave $k+1$ will eventually return. From assumption ii), since $s \leq sr_k \leq ss_{k+1} \leq \min(se_{k+1}^{(1)}, \ldots, se_{k+1}^{(n)}) \leq sr_{k+1}$, assertion $\mathcal{A}[se_{k+1}^{(1)}, \ldots, se_{k+1}^{(n)} \mid sr_{k+1}]$ holds. Stop condition is met by wave $k+1$, which is the last one. □

4.3 Example of implementation on a virtual ring

Processes C_i are connected according to a virtual ring topology. A wave is materialized by a control message, called a *token*, which makes a complete turn around the ring (a token turn implements an iteration step). One of the processes, say C_α, controls the beginning and the end of each turn. Each process C_i can perform two kinds of events: *send the token* (to the successor on the ring): $E_i(token)$, *receive the token* (from the predecessor on the ring): $R_i(token)$. Predicate new_step is a variable new_step_α belonging to C_α, and for every $i \in X$, predicate $wh(i)$ is implemented by a boolean variable wh_i. The token carries a boolean value, whose value is the conjunction of values $A_i[se^{(i)}]$ evaluated by processes C_i previously visited in the current token turn. Thus, $collected(i)$ corresponds to the fact that value $A_i[se^{(i)}]$ has been taken into account in $token.v$. The variable a_i is used by C_i to record the value of the token when it is received. Finally,

	event *start*	corresponds to	$E_\alpha(token)$
$\forall i$	event beg_visit_i	-id.	$R_i(token)$
$\forall i \neq \alpha$	event end_visit_i	-id.	$E_i(token)$
	event *return*	-id.	end_visit_α (internal event)

Thus, behaviour of C_α and of other C_i's slightly differs in the following way: all the controllers, after receiving the token (event beg_visit_i), keep it until their local guard I_i holds, whereafter the C_i's ($i \neq \alpha$) send it to their successor (event end_visit_i, $i \neq \alpha$) whereas C_α evaluates new_step, according to the value of stop predicate NSDE $a_\alpha = \mathcal{A}[se^{(1)}, \ldots, se^{(n)} \mid sr]$ (event $end\_visit_\alpha = return$ then either starts a new step (event $start$) or terminates.

Behaviour rules of these events are given below:

	$\{new\_step_\alpha\}$	$start = E_\alpha(token)$	$\{\neg new\_step_\alpha \wedge \neg wh_\alpha \wedge token.v = A_\alpha\}$
$\forall i$	$\{\neg wh_i\}$	$beg\_visit_i = R_i(token)$	$\{wh_i \wedge a_i = token.v\}$
$\forall i \neq \alpha$	$\{wh_i \wedge I_i\}$	$end\_visit_i = E_i(token)$	$\{\neg wh_i \wedge (token.v = a_i \wedge A_i)\}$
	$\{I_\alpha \wedge wh_\alpha\}$	$return = end\_visit_\alpha$	$\{I \wedge a_\alpha = a \wedge new\_step_\alpha = \neg a_\alpha\}$

It is easy to see that, if channels are reliable and run is fair, this implementation gives raise to live waves.

4.4 Summary

Up to now, we have derived a control program allowing to compute a sequence of NSDEs, based upon the *sequence of guarded waves* iterative scheme. This means that progression in the execution of each wave is submitted to a conjunction of conditions (guards): a wave has to "wait" on each C_i until a guard I_i becomes true; moreover, the stop condition results from a NSDE of local predicates conjunction, each predicate being atomically evaluated by each C_i in a state where I_i is true. This is the general part. There is a particular part when concerned with a detection problem as stated at the beginning of this section: it remains to obtain a NSD-evaluable predicate $\mathcal{F}$ associated to the particuliar predicate $\mathcal{T}$ we want to detect, insuring safety and liveness requirements. In the next section, we solve this problem for the particular example of distributed termination detection. Then we outline how this could be generalized to other kinds of stable predicates.

5 Example: termination detection

5.1 The distributed termination detection problem

Consider a distributed program, called the *underlying program*. The aim of distributed termination detection is to design a (distributed) superimposed control program, detecting the termination of the underlying program. This problem has been put forward by Francez [6] and addressed by many authors [4, 5, 13, 14, 8, 12], resulting in a number of termination detection algorithms. Nevertheless, to our knowledge, none of them used a derivation approach similar to the one presented here. In particular, concepts such as *NSDE* - with *safety* and *liveness* -, *guarded waves* as support for an iterative scheme were not explicitely stated and used to meet the required specifications. Moreover, due to the generality of the guarded wave sequence scheme, our result is a family of detection programs, each particular implementation of the abstract scheme giving a particular program member of this family (some of the previously known algorithms appear to be members of this family).

5.2 Superimposition for termination detection

The role of the control program is to detect the termination of the underlying program. To this end, the former is designed as a superimposed observation program, as explained in preceeding sections. Each process P_i of the underlying program can be *active* or *passive*. This status of P_i is observed by C_i, which can atomically and in any state evaluate a predicate $passive(i)$, whose value is true if, and only if, P_i is passive. The behaviour of control program, with respect to observation of the underlying program, obeys the following rules, for any channel c_{ij} and any message $m \in UM$:

(E) $\{\neg passive(i) \wedge \neg sent(i,j,m)\}$ $E_{ij}(m)$ $\{\neg passive(i) \wedge sent(i,j,m) \wedge \neg rec(i,j,m)\}$
(R) $\{sent(j,i,m) \wedge \neg rec(j,i,m)\}$ $R_{ji}(m)$ $\{\neg passive(i) \wedge rec(j,i,m)\}$
(I) $\{\neg passive(i)\}$ INT_i $\{\neg passive(i) \vee passive(i)\}$

(INT_i is an internal event of C_i corresponding to an internal event of P_i)

rule (E) expresses the fact that only active processes can send underlying messages,

rule (R) expresses the fact that the transition from *passive* to *active* occurs upon the receipt of an underlying message, and only then,

rule (I) expresses the fact that transition from *active* to *passive* is possible in any active state (and corresponds to an underlying internal event) and that underlying internal events are enabled only in active states.

5.3 Specification of termination

Definition We will say that a program has terminated in a state s if, and only if, no event belonging to this program is enabled in this state.

This definition is consistent with the computational model adopted in this paper (§2).

Claim 1 The underlying program has terminated in a state s if, and only if, the following assertions hold in the corresponding state of the control algorithm:

i) $\forall i \in X$: $passive(i)$ (all underlying processes are passive)

ii) $\forall (i,j) \in \Gamma$, $\forall m$: $(m \in UM \wedge sent(i,j,m)) \Rightarrow rec(m,i,j)$ (all underlying channels are "empty")

Proof Suppose the program terminated. Neither underlying *internal* nor underlying message *send* event is enabled, and thus, from rules (E) and (I), $\forall i \in X$: $passive(i)$. No underlying message *receive* event is enabled and thus, from rule (R), $\forall (i,j) \in \Gamma$, $\forall m$: $(m \in UM \wedge sent(i,j,m) \Rightarrow rec(m,i,j))$.

Conversely, from i) follows that no underlying internal and no underlying message *send* event is enabled in state s and ii) expresses the fact that no underlying message *receive* event is enabled in state s ("channels are empty") □

In the rest of this paper, we will use the notation

$$empty(i,j) \equiv \forall m \ (m \in UM \wedge sent(i,j,m)) \Rightarrow rec(m,i,j)$$

From claim 1 we can deduce that the control program has to detect a state in which the following predicate holds:

$$terminated \equiv \bigwedge_{i \in X} passive(i) \wedge \bigwedge_{(i,j) \in \Gamma} empty(i,j)$$

Claim 2 In any run of the control program, *terminated* is a stable predicate.

Proof We have to show: $terminated[s] \wedge s' > s \Rightarrow terminated[s']$. In fact, when $terminated[s]$ holds, it results from claim 1 that no underlying message *send* or *receive* event is enabled in state s. Now, only an underlying message *receive* event could set to false a predicate $passive(i)$, and only an underlying message *send* event could set to false a predicate $empty(i,j)$. Thus, *terminated* remains true in all control program states subsequent to s □.

5.4 Derivation of a predicate $\mathcal{F}$

Predicate *terminated* cannot be atomically evaluated by any of the control processes. More precisely:

1. Neither C_i nor C_j can evaluate atomically the predicate $empty(i,j)$
2. Even if we assume that $empty(i,j)$ could be atomically evaluated by a control process, or if transmissions were instantaneous, it remains that no control process can evaluate simultaneously all the predicates $passive(i)$ and $empty(i,j)$; this is why we need the notion of $NSDE$ of a predicate.

The first point is easy to overcome. Predicate $empty(i,j)$ can be evaluated with delay by C_i, if we associate to each underlying message m sent on channel c_{ij} an event of C_i, say $ack_i(m)$, such that $E_{ij}(m) \stackrel{*}{\Rightarrow} ack_i(m)$ (liveness), and $en(ack_i(m)) \Rightarrow rec(i,j,m)$ (safety). Such an event corresponds to the fact that C_i becomes aware of the occurrence of event $R_{ij}(m)$; it can be, for example, the receipt of an acknowledgment message (control message) sent back by C_j to C_i, or else the reach of a maximum delivery time by a clock local to C_i (control internal event), etc. If we denote by $acknowledged(i,j,m)$ the occurrence predicate of this event, the assertion $acknowledged(i,j,m) \Rightarrow rec(i,j,m)$ holds in any state. Hence, if we set

$$a\_empty(i) \equiv \bigwedge_{j \in \Gamma(i)} \forall m \ (m \in UM \wedge sent(i,j,m) \Rightarrow acknowledged(i,j,m))$$

(where $\Gamma(i) = \{j \mid j \in X \wedge (i,j) \in \Gamma\}$) we have, in any state s: $a\_empty(i) \Rightarrow \bigwedge_{j \in \Gamma(i)} empty(i,j)$ and thus

$$\bigwedge_{i \in X} (passive(i) \wedge a\_empty(i)) \Rightarrow terminated$$

The second point is more intricate: it constitutes the core of the difficulty for termination detection in an asynchronous distributed context. In fact, it is required to determine a predicate $\mathcal{F}$, *NSD-evaluable*, strong enough to insure safety but weak enough to insure liveness. For that purpose, let us analyse more accurately the predicate *terminated* to be detected by $\mathcal{F}$.

Claim 3 Given a state s, the assertion *terminated*$[s]$ holds if, and only if, a n-uple of states $(qs^{(1)}, \ldots, qs^{(n)})$ exists, for which the following assertions are satisfied:

i) $\max(qs^{(1)}, \ldots, qs^{(n)}) \leq s$ and

ii) $\forall i \in X \ : \ qs^{(i)} \leq \sigma \leq s \Rightarrow passive(i)[\sigma]$ and

iii) $\forall i \in X \ : \ \left(\bigwedge_{j \in \Gamma(i)} empty(i,j)\right)[qs^{(i)}]$

Proof Suppose *terminated*$[s]$. Hence, $\forall i \in X \ : \ passive(i)[s]$. The set of states $\{\underline{s} \mid \underline{s} \leq \sigma \leq s \Rightarrow passive(i)[\sigma]\}$ is non-empty and thus has a minimum element, say $ps^{(i)}$. Similarly, $\forall i \in X \ : \ \left(\bigwedge_{j \in \Gamma(i)} empty(i,j)\right)[s]$, hence the set of states $\{\underline{s} \mid \underline{s} \leq \sigma \leq s \Rightarrow \left(\bigwedge_{j \in \Gamma(i)} empty(i,j)\right)[\sigma]\}$ is non-empty and has a minimum element, say $cs^{(i)}$. Clearly, the states $qs^{(i)} = \max(ps^{(i)}, cs^{(i)})$ $(i \in X)$ satisfy assertions i), ii), iii).

Conversely, suppose i),ii),iii) hold. From behaviour rule (E) and assertion ii), no event $E_{ij}(m)$, where $(i,j) \in \Gamma$, $m \in UM$, occurs in states σ such that $qs^{(i)} \leq \sigma \leq s$ whence, from assertion i): $\forall (i,j) \in \Gamma \ : \ empty(i,j)[s]$. Moreover, from assertion ii) again, $\forall i \in X \ : \ passive(i)[s]$. Thus *terminated*$[s]$. □

This characterization means that *terminated*$[s]$ holds if, and only if, each controller C_i has observed that P_i remained *continuously passive* since a state $qs^{(i)}$ prior to s. Obviously, *terminated* being stable (claim 2), assertions *passive*(i) will hold continuously after state s too.

Recall that our detection is based upon a sequence of NSDEs $\max(se_k^{(1)}, \ldots, se_k^{(n)}) \leq sr_k \leq \min(se_{k+1}^{(1)}, \ldots, se_{k+1}^{(n)})$, $(k = 0, 1, \ldots)$ such that C_i's contribution to the k^{th} NSDE is made in state $se_k^{(i)}$, and global result is recorded in state sr_k. From claim 3, *terminated*$[sr_k]$ cannot be asserted, even if it does hold, unless $\forall i \in X \ : \ qs^{(i)} \leq se_k^{(i)}$. Moreover, the smallest integer k such that $\forall i \in X \ : \ qs^{(i)} \leq se_k^{(i)}$ is characterized by $\forall i \in X \ : \ se_k^{(i)} \leq \sigma \leq sr_k \Rightarrow passive(i)[\sigma]$. But, as we just pointed out, consecutive C_i's contributions are made in states $\ldots, se_k^{(i)}, se_{k+1}^{(i)}, \ldots$ (C_i is not aware of state sr_k). Thus the smallest integer k such that, $\forall i \in X \ : \ qs^{(i)} \leq se_k^{(i)}$, is locally detected on each C_i by $se_k^{(i)} \leq \sigma \leq se_{k+1}^{(i)} \Rightarrow passive(i)[\sigma]$.

Introducing the predicate

$$cp(i,s)[s'] \equiv ((s' \geq s) \wedge (s \leq \sigma \leq s' \Rightarrow passive(i)[\sigma]))$$

(in other words: $cp(i,s)[\sigma]$ holds if, and only if, C_i observed that P_i remained continuously passive between states s and σ), the above analysis shows that the predicate whose

NSDE has to be performed by the k^{th} wave must take into account, on the one hand, the continuous passivity of C_i's between state se^i_{k-1} and state se^i_k, and on the other hand, emptyness of C_i's output channels in state se^i_{k-1}. Thus, if we define, for any $k > 0$:

$$\text{in any state } s\ \mathcal{F}_{k-1}[s] \equiv \bigwedge_{i \in X} (cp(i, se^{(i)}_{k-1})[s] \wedge a\_empty(i)[se^{(i)}_{k-1}])$$

the k^{th} wave can perform a NSDE of $\mathcal{F}_{k-1}$.

Proposition 1 Every NSDE $\mathcal{F}_{k-1}[se^{(1)}_k, \ldots, se^{(n)}_k \mid sr_k]$ is safe with respect to *terminated*.

Proof Suppose $\mathcal{F}_{k-1}[se^{(1)}_k, \ldots, se^{(n)}_k \mid sr_k]$; this implies

$$\bigwedge_{i \in X} (cp(i, se^{(i)}_{k-1})[se^{(i)}_k] \wedge a\_empty(i)[se^{(i)}_{k-1}])$$

and thus, from the definition of *a_empty*:

$$\bigwedge_{i \in X} (cp(i, se^{(i)}_{k-1})[se^{(i)}_k] \wedge \bigwedge_{j \in \Gamma(i)} empty(i,j)[se^{(i)}_{k-1}])$$

Since $\forall i \in X\ :\ se^{(i)}_{k-1} \leq sr_{k-1} \leq se^{(i)}_k$, sufficient conditions (i), (ii), (iii) of claim 3 hold, with $qs^{(i)} = se^{(i)}_{k-1}$ and $s = sr_{k-1}$, whence $terminated[sr_{k-1}]$. Since $sr_{k-1} \leq sr_k$ we have, by claim 2: $terminated[sr_k]$. □

This proposition shows that a detection can be based upon a sequence of NSDEs of predicates $\mathcal{F}_k$. It remains to split these predicates into an invariant and a stop condition. By definition,

$$\begin{aligned}\mathcal{F}_{k-1} &\equiv \bigwedge_{i \in X} cp(i, se^{(i)}_{k-1}) \wedge \bigwedge_{i \in X} (a\_empty(i)[se^{(i)}_{k-1}]) \\ &\equiv \bigwedge_{i \in X} cp(i, se^{(i)}_{k-1}) \wedge \bigwedge_{i \in X} (passive(i) \wedge a\_empty(i))[se^{(i)}_{k-1}]\end{aligned}$$

One of the two terms can be taken as an invariant, and the other as a stop condition. Recall that the invariant is a conjunction of assertions I_i, atomically evaluable by C_i, such that $\forall k\ :\ I_i[se^{(i)}_k]$ is true. Since it's not possible to assert that an underlying process remains continuously passive between two consecutive *end_visit* events, assertions $cp(i, se^{(i)}_{k-1})[se^{(i)}_k]$ don't meet the invariance requirements. Thus, we choose

- as the invariant I which governs the progress of the wave:

$$\bigwedge_{i \in X} I_i[se^{(i)}_k] \equiv \bigwedge_{i \in X} (passive(i) \wedge a\_empty(i))[se^{(i)}_k]$$

- as the stop condition $\mathcal{A}[se^{(1)}_k, \ldots, se^{(n)}_k \mid sr_k]$ the value

$$\bigwedge_{i \in X} cp(i, se^{(i)}_{k-1})[se^{(i)}_k]$$

which is obtained in state sr_k as the result of the wave k .

Proposition 2 The detection of predicate *terminated* is live.

Proof Let s be a state such that *terminated* holds. No event $E_{ij}(m)$, $(i,j) \in \Gamma$, $m \in UM$ is enabled in states $\sigma \geq s$ and thus, for each $i \in X$, the state in which last event $E_{ij}(m)$ occurred is well-defined (by convention, it is the initial state s_0 if C_i sent no underlying message). Since $E_{ij}(m) \stackrel{*}{\Rightarrow} ack_i(m)$, the state in which last event $ack_i(m)$ occurred is well-defined : let $la^{(i)}$ denote this state (by convention, $la^{(i)} = s_0$ if C_i sent no underlying message). In any state $\sigma \geq la^{(i)}$, assertion $a\_empty[\sigma]$ holds and in any state $\sigma \geq s$, assertion $passive(i)[\sigma]$ holds. Thus, the first assumption of lemma 4.1 (in §4.2) is verified, with $\sigma^{(i)} = \max(la^{(i)}, s)$.

On the other hand, from claim 3, we have $\sigma \geq s \Rightarrow cp(i,s)[\sigma]$ and thus the second assumption of lemma 4.1 is verified too. Thus, by lemma 4.1, detection is live. □

5.5 Specification of a control program for termination detection

In this subsection, we summarize the results derived up to now. In order to make a new step down to a more concrete specification, we implement with variables some of the predicates used in behaviour rules. Predicates $passive(i)$, $wh(i)$, new_step are simply implemented by boolean variables $passive_i$, wh_i, new_step. Predicates $cp(i,s)$ are implemented by boolean variables cp_i, such that cp_i has the value *true* in the post-condition of event end_visit_i, and becomes false as soon as $passive_i$ becomes false. Predicates $a\_empty(i)$ can be implemented by integer variables; in fact, recall that

$$a\_empty(i) \equiv \bigwedge_{j \in \Gamma(i)} \forall m \ (m \in UM \wedge sent(i,j,m) \Rightarrow acknowledged(i,j,m))$$

If we set $\#sent_i = \sum_{j \in \Gamma(i)} \text{card}\{m \mid m \in UM \wedge sent(i,j,m)\}$
$\#ack_i = \sum_{j \in \Gamma(i)} \text{card}\{m \mid m \in UM \wedge acknowledged(i,j,m)\}$

we have, in any state: $\#sent_i \geq \#ack_i$ and $a\_empty(i) \equiv (\#sent_i = \#ack_i)$. Thus, if we let $nack_i = \#sent_i - \#ack_i$, we have, in any state, $nack_i \geq 0$ and $a\_empty(i) \equiv (nack_i = 0)$. For each $i \in X$, $j \in \Gamma(i)$, $m \in UM$, integer variable $nack_i$ is involved in behaviour rules of $E_{ij}(m)$ and $ack_i(m)$.

$\{\neg passive_i \wedge nack_i = \alpha \wedge \alpha \geq 0\}$	$E_{ij}(m)$	$\{\neg passive_i \wedge nack_i = \alpha + 1\}$
$\{\text{true}\}$	$R_{ji}(m)$	$\{\neg passive_i \wedge \neg cp_i\}$
$\{\neg passive_i\}$	INT_i	$\{\neg passive_i \vee passive_i\}$
$\{nack_i = \alpha \wedge \alpha > 0\}$	$ack_i(m)$	$\{nack_i = \alpha - 1\}$
$\{new\_step\}$	$start$	$\{\neg new\_step \wedge \bigwedge_{i \in X} (\neg collected(i))\}$
$\{\neg wh_i \wedge \neg collected(i)\}$	beg_visit_i	$\{wh_i \wedge \neg collected(i)\}$
$\{wh_i \wedge \neg collected(i) \wedge passive_i \wedge (nack_i = 0)\}$		
	end_visit_i	$\{\neg wh_i \wedge collected(i) \wedge cp_i\}$
$\{\bigwedge_{i \in X} (passive_i \wedge (nack_i = 0)[se^{(i)}] \wedge collected(i))\}$		
	$return$	$\{a = \bigwedge_{i \in X} cp_i[se^{(i)}] \wedge new\_step = \neg a\}$

As can be seen, predicates $sent(i,j,m)$ and $rec(i,j,m)$ are not implemented by control variables (actually, they could be implemented by underlying variables). Predicates

collected(i) express the fact that the value $cp_i[se^{(i)}]$ has been taken into account in the current NSDE. How they are implemented depends of the particular implementation of waves (ring with token, rooted or unrooted spanning tree, etc. [9]). In the annex, we give the complete control program, in a CSP-like syntax, with a ring implementation for the waves (see §4.3) and an acknowledgement message implementation for the acknowledgement mechanism. Both insure liveness conditions.

6 Conclusion

Methodological design of distributed programs is of major concern to master parallelism. Distributed detection problems are important, due to their role in distributed systems. Since their solution essentially involve predicate evaluations, we have put forward the difficulties inherent to distributed evaluation. After recalling that, in asynchronous distributed context, such evaluations can be only non-simultaneous and delayed, we have formalized this concept as the NSDE paradigm, and we have given precise definitions of its associated safety and liveness properties. The detection problem itself has been set in its generality such that, whatever the predicate to detect, the solution is expressed by an iterative NSDE scheme. The well-known technique used in centralized context, which consists in decomposition of a goal specification into an invariant and a stop condition, has been extended to the asynchronous distributed context: this leads to a "two-level" iterative scheme, namely a spatial iteration (*for all processes do ...*), which is a NSDE implemented by a wave, and a temporal iteration (*while* $\neg$ *stop do* NSDE *done*), implemented by a sequence of waves. The systematic design of the wave (a step of the temporal iteration) is deduced from its specification: the concept of guarded wave is introduced, in order to insure the desired invariant (the guards control the wave progression), and the stop condition is evaluated by the non simultaneous delayed evaluation scheme realized by the spatial iteration.

As an example illustrating this general technique, we have addressed the problem of distributed termination detection. The point here is to obtain a NSDE-evaluable predicate associated with the particuliar predicate which must be detected, while meeting safety and liveness properties. To show the generality of this approach, we outline how it could be applied to other kinds of stable predicates, e.g. those like $T[s] \equiv (G[s] \geq \alpha)$, where G is a monotonic non-decreasing function defined over the states of a distributed program, valued in a semi-lower lattice E, and α is a particular value in E (threshold). Suppose G is of the following form: each process P_i of the program records a value $x_i \in E$, and each message m carries a value $m.v \in E$. In any state s of the program, $G[s] = \inf(\inf(x_i \mid i \in X), \inf(m.v \mid m \textit{ is in transit in } s))$, where inf denotes the infimum operation on E and m in transit in s means that $sent(.,.,m) \wedge \neg received(.,.,m)$ holds in state s. If the program obeys the following behaviour rules, it can be shown that G is monotonic non-decreasing ([17][9, chapter 2]):

1. (emission) when P_i sends m, them $m.v = x_i$
2. (reception) when P_i receives m, then x_i records a value v', with $v' \geq \inf(x_i, m.v)$
3. (internal event) P_i can spontaneously decrease its recordrd value x_i

A derivation similar to the one done in §5 above leads to NSD-evaluable predicates defined (informally) by: *for every $i \in X$, all the values recorded by P_i since the last visit have been $\geq \alpha$* **and** *every message m sent by P_i with a value $m.v < \alpha$ has been received.*

Termination is a particular case: this is obvious with $E = \{active, passive\}$, $active \leq passive$, $\alpha = passive$. Another interesting example is the calculus of Global Virtual Time (GVT) of a distributed simulation [11]: the GVT is the smallest simulation time reached by local simulators or carried by in-transit messages. When the GVT is greater than a value α, this means that the overall simulation time has reached this value.

Acknowledgements: We would like to thank the anonymous referees for their useful comments which helped us to obtain very significant improvements.

References

[1] L. Bougé and N. Francez. A Compositional Approach to Superimposition. in *Proc. ACM Symposium on POPL*, San Diego, 1988.

[2] K.M. Chandy and L. Lamport. Distributed Snapshots : Determining Global States of Distributed Systems. *ACM TOCS*, 3(1):63–75, Feb. 1985.

[3] K.M. Chandy and J. Misra. *Parallel Program Design: a Foundation.* Addison Wesley, 1988.

[4] E.W. Dijkstra and C.S. Scholten. Termination Detection for Diffusing Computations. *Inf. Proccessing Letters*, Vol. 11:1–4, 1980.

[5] E.W. Dijkstra, W. H. J. Feijen, and A. J. M. Van Gasteren. Derivation of Termination Detection Algorithm for Distributed Computation. *Inf. Processing Letters*, Vol. 16:217–219, June 1983.

[6] N. Francez. Distributed Termination. *ACM Toplas*, 2-1, 1980.

[7] J.M. Hélary. Observing Global States of Asynchronous Distributed Computations. in *Proc. 3rd Int. Workshop on Distributed Algorithms*, Nice, sept. 1989. Springer-Verlag LNCS 392:124-135.

[8] J. M Hélary, C. Jard, N. Plouzeau, and M. Raynal. Detection of Stable Properties in Distributed Applications. in *Proc. 6th annual ACM Symposium on Principles of Distributed Computing*, pages 125–136, Vancouver, August 1987.

[9] J. M. Hélary, M. Raynal. *Control and Synchronisation of Distributed Systems and Programs.* Wiley Series in Parallel Computing, August 1990.

[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prenctice-Hall, 1984.

[11] D. Jefferson. Virtual Time. *ACM Toplas*, 7(3):404-425, July 1985.

[12] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2:161–175, 1987.

[13] J. Misra. Detecting Termination of Distributed Computation Using Markers. in *Proc. 2d ACM Symp. on PODC*, Montreal, 1983, pp 290-294.

[14] N. Shavit and N. Francez. A New Approach to Detection of Locally Indicative Stability. in *Proc. 13th ICALP*, LNCS, 226, pages 344-358, Springer-Verlag, 1986.

[15] F.P. Schneider. Paradigms for Distributed Programs. In *Distributed Systems*, LNCS 190: 431-480, Springer-Verlag Ed., 1985.

[16] A. Segall. Distributed Network Protocols. *IEEE Trans. on Inf. Theory*, IT 29(1):23–35, jan. 1983.

[17] G. Tel. Distributed Infimum Approximation. in *Proc. FTC 87*, LNCS 278, Bordach et al ed., 440-447, 1987.

Annex

We give a completely specified program for termination detection, where, in addition to the previously implemented predicates (§5.5), waves are implemented on a virtual ring (§4.3) and acknowledgement mechanism is implemented by acknowledgement messages (each *ack* event corresponds to the receipt of an acknowledgement message, and the receipt of an underlying message triggers the send of the corresponding acknowledgement message).

The program is given in a CSP-like syntax ([10]), adapted to the asynchronous model described in section 2 (sends and receipts are denoted respectively by !! and ??). Each behaviour rule is translated into a guarded command, according to the following rules:

- $\{p\}\ E_{ij}(m)\ \{q\}$ is translated into a guarded command for process i: $p \rightarrow C_j$!!m; S where S is a sequential program such that $\{p\}\ S\ \{q\}$ (in the usual meaning),
- $\{p\}\ R_{ji}(m)\ \{q\}$ is translated into a guarded command for process i: $p \wedge C_j??m \rightarrow$ S where S is a sequential program such that $\{p\}\ S\ \{q\}$,
- $\{p\}\ INT_i\ \{q\}$ is translated into a guarded command for process i: $p \rightarrow$ S where S is a sequential program such that $\{p\}\ S\ \{q\}$.

A program is then a collection of sequential programs for each process, having the structure:

C_i:: init; %*init assigns initial values to variables; stop is initialized to false*%
  * [¬stop →
  ⋮
  ]

The purpose of the variable *stop* is to control the end of the process sequential program. A message *terminate* is sent around the ring by C_α when the latter detects underlying programm termination. Its purpose is only to inform other controllers of this detection and to stop their local program.

Comments are bracketed with %.

Text of the program

C_α::new_step:=true; nack:=0; cp:=true; stop:=false;
* [¬stop →
[new_step →C_{succ}!!token(cp);wh:=false;cp:=true;new_step:=false
□
¬wh ∧ C_{pred}??token(v) → a:=v; wh:=true
□
wh ∧ passive ∧ nack=0 →new_step:=¬a;
[
¬new_step → %termination detected%
C_{succ}!!terminate
]
□
%P_α *receives m from* P_j% C_j??m → passive:=false; cp:=false; C_j!!ack
□
¬passive %∧P_α *sends m to* P_j% → C_j!!m; nack:=nack+1
□
C_j??ack → nack:=nack-1
□
¬passive %∧P_α *becomes passive*% → passive:=true
□
C_{pred}??terminate → stop:=true
]
]

$C_i(i \neq \alpha)$::nack:=0; cp:=true; wh:=false;stop:=false
* [¬*stop* →
[¬wh ∧ C_{pred}??token(v) → a:=v; wh:=true
□
wh ∧ passive ∧ nack=0 → C_{succ}!!token(a ∧ cp);wh:=false;cp:=true
□
%P_i *receives m from* P_j% C_j??m → passive:=false; cp:=false; C_j!!ack
□
¬passive %∧P_i *sends m to* P_j% → C_j!!m; nack:=nack+1
□
C_j??ack → nack:=nack-1
□
¬passive %∧P_i *becomes passive*% → passive:=true
□
C_{pred}??terminate → C_{succ}!!terminate; stop:=true
]
]

LISTE DES DERNIERES PUBLICATIONS INTERNES IRISA
1991

PI 580 DESCRIPTION ET SIMULATION D'UN SYSTEME DE CONTROLE DE PASSAGE A NIVEAU EN SIGNAL
Bruno DUTERTRE
Mars 1991, 66 Pages.

PI 581 THE SYNCHRONOUS APPROACH TO REACTIVE AND REAL-TIME SYSTEMS
Albert BENVENISTE
Avril 1991, 36 Pages.

PI 582 PROGRAMMING REAL TIME APPLICATIONS WITH SIGNAL
Paul LE GUERNIC, Michel LE BORGNE, Thierry GAUTIER
Claude LE MAIRE
Avril 1991, 36 Pages.

PI 583 ELIMINATION OF REDUNDANCY FROM FUNCTIONS DEFINED BY SCHEMES
Didier CAUCAL
Avril 1991, 22 Pages.

PI 584 TECHNIQUES POUR LA MISE AU POINT DE PROGRAMMES REPARTIS
Michel ADAM, Michel HURFIN, Michel RAYNAL, Noël PLOUZEAU
Mai 1991, 10 Pages.

PI 585 TOWARDS THE CONSTRUCTION OF DISTRIBUTED DETECTION PROGRAMS, WITH AN APPLICATION TO DISTRIBUTED TERMINATION
Jean-Michel HELARY
Mai 1991, 24 pages.

PI 586 OPAC : A COST-EFFECTIVE FOATING-POINT COPROCESSOR
André SEZNEC, Karl COURTEL
Mai 1991, 26 Pages.

PI 587 ON FAILURE DETECTION AND IDENTIFICATION : AN OPTIMUM ROBUST MIN-MAX APPROACH
Elias WAHNON, Albert BENVENISTE
Mai 1991, 24 Pages.

PI 588 BOUNDED-MEMORY ALGORITHMS FOR VERIFICATION ON THE FLY
Claude JARD, Thierry JERON
Mai 1991, 14 pages.

ISSN 0249-6399